ANL/ALCF-17/3

Evaluation of the OpenCL AES Kernel Using the Intel FPGA SDK for OpenCL

Argonne Leadership Computing Facility

ANL/ALCF-17/3

Evaluation of the OpenCL AES Kernel Using the Intel FPGA SDK for OpenCL

prepared by
Zheming Jin, Kazutomo Yoshii, Hal Finkel, Franck Cappello

Argonne Leadership Computing Facility, Argonne National Laboratory

April 20, 2017

Evaluation of the OpenCL AES Kernel using the Intel FPGA SDK for OpenCL

Introduction

The OpenCL standard is an open programming model for accelerating algorithms on heterogeneous computing system. OpenCL extends the C-based programming language for developing portable codes on different platforms such as CPU, Graphics processing units (GPUs), Digital Signal Processors (DSPs) and Field Programmable Gate Arrays (FPGAs). The Intel FPGA SDK for OpenCL is a suite of tools that allows developers to abstract away the complex FPGA-based development flow for a high-level software development flow. Users can focus on the design of hardware-accelerated kernel functions in OpenCL and then direct the tools to generate the low-level FPGA implementations. The approach makes the FPGA-based development more accessible to software users as the needs for hybrid computing using CPUs and FPGAs are increasing. It can also significantly reduce the hardware development time as users can evaluate different ideas with high-level language without deep FPGA domain knowledge.

The Advanced Encryption Standard (AES) specifies a standard encryption algorithm that is used worldwide to protect electronic data. The OpenCL AES kernel was originally developed by Liu et al. [1] at Virginia Tech. The kernel function converts data to an unintelligible form using cryptographic keys. The kernel is constructed as a dynamic library engine that can be linked into the OpenSSL framework. The authors evaluated the performance of the kernel on the scalable multi-FPGA architecture [2]. Their hardware platform is based on the M506 module with a StratixV A3 FPGA and 8GB DDR3 memory. On a single M506 module, they achieved the highest FPGA throughput of 5.1Gbps (Giga bits per second) using the OpenCL SIMD4 vectorization optimization. The throughput of the same test is 4.4Gbps on an i7-4770K processor.

In this report, we evaluate the performance of the kernel using the Intel FPGA SDK for OpenCL and Nallatech 385A FPGA board. Compared to the M506 module, the board provides more hardware resources for a larger design exploration space. The kernel performance is measured with the compute kernel throughput, an upper bound to the FPGA throughput. The report presents the experimental results in details. The Appendix lists the kernel source code.

Overview of the two FPGA devices

FPGA offers a wide variety of configurable memories, high-speed I/Os, logic blocks and routing. StratixV and Arria10 series of Intel FPGAs are two products for high-performance applications. Table 1 compares the technology and resource counts of the StratixV A3 FPGA device [3] on the M506 with those of the Arria10 GX1150 FPGA device [4] on the Nallatech 385A. The Arria10 device features 20-nm SoC process technology and operates at 0.95 V core voltage while the StratixV device uses 28-nm technology and operates at 0.9 V. Based on the results in Table 1, the Arria10 device has approximately 3X more resources than the StratixV device, so it provides a larger design exploration space for the performance evaluation of the AES kernels.

Table 1. Device overview of two FPGAs

Device	Technology	Logic elements	ALMs	Register	M20K memory bits
Stratix 5SGXMA3	28nm	340K	128,300	513K	19Mb
Arria 10AX115	20nm	1150K	427,200	1708800	54260Kb

Nallatech 385A

Nallatech 385A is a PCIe-based FPGA accelerator card that features an Arria 10 GX1150 FPGA device, PCIe × 8 Generation 3 host interface, and two banks of 4GB DDR3 memory. The theoretical peak floating-point performance is 1.5 TFLOPS and the theoretical peak memory bandwidth approximately 34 GB/s.

Kernel optimizations

As described in [5], users can take advantage of compute unit replication and kernel SIMD vectorization to achieve higher throughput or lower kernel time. The compute device replication generates multiple compute units for each kernel. Each compute unit has its own memory access interface. The SIMD vectorization duplicates only the data path of the compute unit without generating additional memory interfaces. When the kernel is vectorized, the static memory coalescing is performed automatically by the compiler to generate a memory interface that can coalesce the multiple memory loads into a single wide load. While there is no limit to the number of kernel copies that users can specify, the number of SIMD lanes must be a power of two. The compiler will give a warning when the width of all the lanes exceeds the memory interface data width.

Experimental setup

In this work, a host system is set up with two 2.6 GHz Intel Xeon processors and 32GB DDR3 memory for each node. The PCI Express provides a Gen2×8 connection. CentOS 6.8 with Linux kernel 2.6.32 is installed as the operating system. We used the Intel's FPGA SDK for OpenCL version 16.0.2 Pro Prime for the experimental results.

For the kernel test, we choose the same input data size as in the paper [2]. The size of the input data is 2GB and the block size is 128MB. The host program divides the input data into 16 blocks (2GB/128MB) and they are encrypted sequentially in 16 passes. The test mode is the AES 256-bit algorithm in ECB mode.

Experimental results

Table 2 lists the FPGA resource usage and the maximum frequency (Fmax) of each kernel. The *default* kernel is the baseline kernel without any kernel optimization.

Replication of compute unit is represented as "cuX" where X indicates the replication times. The combination of kernel duplication and N-lane vectorization is represented as "simdN+cuX". The logic utilization of each kernel is below 40%. The RAM block usage increases from 17% (simd2) to 45% (simd16) for the kernel vectorization and from 20% (cu2) to 67% (cu16) for the kernel duplication. The significant increase in RAM blocks for both optimizations is due to the duplication of the four 256-entry by 32-bit look-up tables in the OpenCL AES kernel. The SDK fails to build when there are more than two duplicate kernels with SIMD16 vectorization, because they require more RAM blocks than the device can provide. As shown in the table, kernel duplication degrades the Fmax from 247 MHz to 186 MHz while kernel vectorization decreases the maximum frequency from 250 MHz to 218 MHz. This indicates the impact of increasing RAM blocks on the timing of the kernel implementation.

Table 2. FPGA resource usage and Fmax of the implemented kernels

Kernel	Logic Utilization	Memory bits	RAM blocks	Fmax (MHz)
default	13%	8%	16%	245
simd2	13%	8%	17%	249
simd4	14%	9%	21%	250
simd8	15%	10%	29%	238
simd16	18%	14%	45%	218
cu2	14%	8%	20%	247
cu4	17%	10%	26%	233
cu8	24%	12%	40%	200
cu16	36%	17%	67%	186
simd16+cu2	25%	21%	78%	207

The FPGA power consumption fluctuates between 30W and 60W based on the power meters reading at 100ms interval over the time of testing all the kernels. We use the compute kernel throughput (Mbps) to measure the performance of each implementation. The compute kernel throughput is calculated by dividing the bit size of the workload over the total kernel execution time on the FPGA. The kernel execution time on an FPGA is consistent. We found the FPGA throughput, which is calculated by dividing the workload

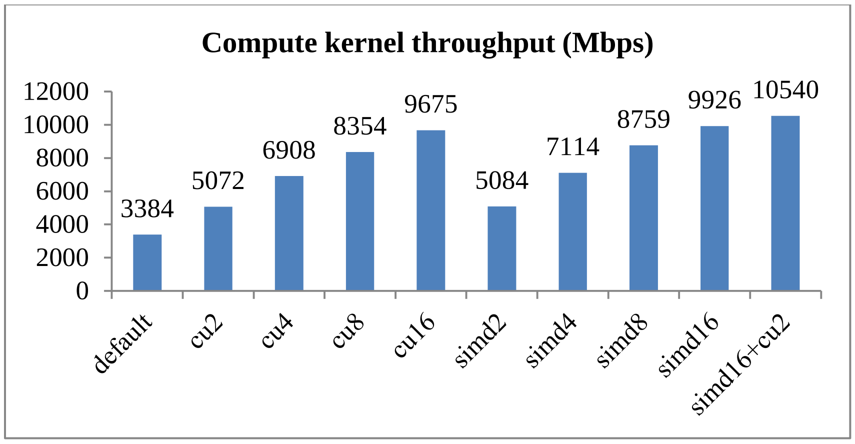

Figure 1 Compute kernel throughput on the Nallatach 385A FPGA board

over the total execution time, is not consistent for each application run. The total execution time includes the data transfer time between the host and the device, kernel execution time and the host execution time. While the compute kernel throughput is the upper bound to the FPGA throughput, it accurately reflects the performance of each kernel running on an FPGA device.

As shown in Figure 1, the kernel throughput does not increase linearly with the increase of the number of compute units and/or vector lanes. The throughput increases by less than 50% when the number of compute units or vector lanes doubles from two to sixteen. The throughput of each SIMD implementations is slightly better than that of compute unit duplications.

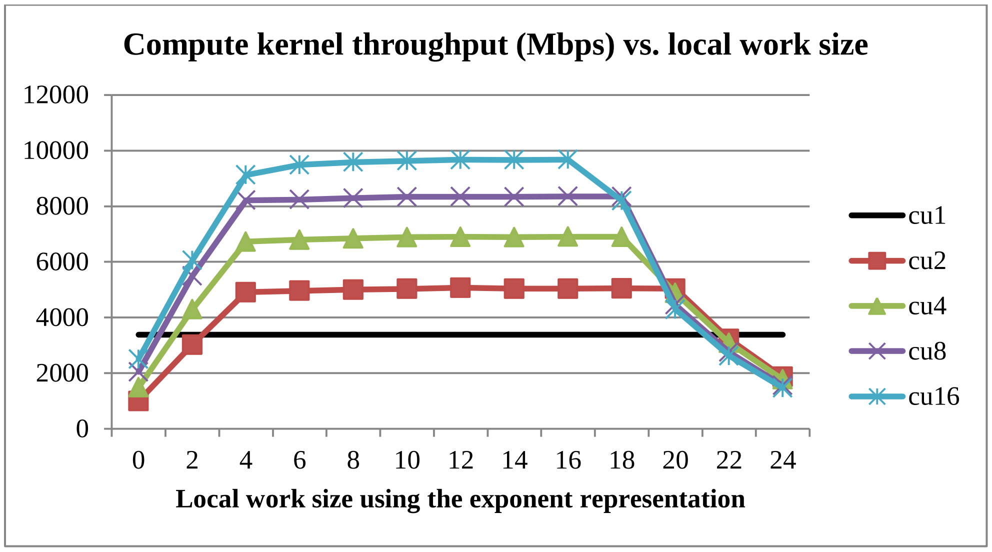

Figure 2 Compute kernel throughput vs. Local work size in kernel duplication

In the kernel duplication optimization, it is interesting that the kernel performance is intimately related with the local work size [6] for multiple compute units. Figure 2 presents the impact of the local work size on the compute kernel throughput. The *x* axis represents the local work size using the exponent as shorthand for $2^{exponent}$. The local work sizes are multiples of four. When there is a single compute unit (as a reference), the kernel throughput slightly fluctuates between 3,382 Mbps and 3,384 Mbps over local work sizes. For multiple compute units, the kernel throughput reaches its maximum when the local work size ranges from 2^{10} to 2^{16}.

Conclusion

This report provides detailed results of the OpenCL AES kernel implemented on a single Arria10 GX1150 FPGA board that is available in the laboratory. In our experiment, the compute kernel throughput is the upper bound to the FPGA throughput for the AES kernel. The results of the compute kernel throughput show the kernel duplication and vectorization improve the kernel performance at the cost of high hardware resources. Using more compute units or vector lanes increases the block RAM usage, which in turn degrades the maximum frequency of the FPGA implementations. The best performance using kernel duplication is achieved by experimenting with the local work sizes.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy Office of Science, under contract DEAC02-06CH11357.

Reference

[1] Z. Liu and A. R. M. Ganesh "OpenCL-AES", Dec. 2011 URL: http://www.github.com/softboysxp/OpenCL-AES.
[2] S. Gao and J. Chritz, "Characterization of OpenCL on a scalable FPGA architecture," *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, Cancun, 2014, pp. 1-6. doi: 10.1109/ReConFig.2014.7032505
[3] Stratix V Device Overview
[4] Arria 10 Device Overview
[5] Intel FPGA SDK for OpenCL. Programming Guide. UG-OCL002. 2016.10.31
[6] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueNDRangeKernel.html

Appendix

```
/**
 * Copyright 2011 University of Virginia. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without modification, are
 * permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice, this list of
 * conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright notice, this list
 * of conditions and the following disclaimer in the documentation and/or other materials
 * provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY <COPYRIGHT HOLDER> ''AS IS'' AND ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
 * FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
 * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
 * ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
 * ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

// The array values omitted to save space
__constant uint Te0[256] = {… …}
__constant uint Te1[256] = {… …}
__constant uint Te2[256] = {… …}
__constant uint Te3[256] = {… …}

__kernel void AES_encrypt (__global uint4 *state, __constant uint4 *rk, uint rounds) {

  uint global_id = get_global_id(0);
  uint4 s, t;
  s = state[global_id] ^ rk[0];

  uint r = rounds >> 1;
  uint4 offset0, offset1, offset2, offset3;
  for (;;) {
    offset0 = s & 0xff;
    offset1 = (s.yzwx >> 8) & 0xff;
    offset2 = (s.zwxy >> 16) & 0xff;
    offset3 = (s.wxyz >> 24);
    t = (uint4)(Te0[offset0.x], Te0[offset0.y], Te0[offset0.z], Te0[offset0.w]) ^
      (uint4)(Te1[offset1.x], Te1[offset1.y], Te1[offset1.z], Te1[offset1.w]) ^
      (uint4)(Te2[offset2.x], Te2[offset2.y], Te2[offset2.z], Te2[offset2.w]) ^
      (uint4)(Te3[offset3.x], Te3[offset3.y], Te3[offset3.z], Te3[offset3.w]) ^
      rk[1];

    rk += 2;
    if (--r == 0) {
      break;
    }

    offset0 = t & 0xff;
    offset1 = (t.yzwx >> 8) & 0xff;
    offset2 = (t.zwxy >> 16) & 0xff;
    offset3 = (t.wxyz >> 24);
    s = (uint4)(Te0[offset0.x], Te0[offset0.y], Te0[offset0.z], Te0[offset0.w]) ^
      (uint4)(Te1[offset1.x], Te1[offset1.y], Te1[offset1.z], Te1[offset1.w]) ^
      (uint4)(Te2[offset2.x], Te2[offset2.y], Te2[offset2.z], Te2[offset2.w]) ^
      (uint4)(Te3[offset3.x], Te3[offset3.y], Te3[offset3.z], Te3[offset3.w]) ^
      rk[0];
  }

  offset0 = (t.zwxy >> 16) & 0xff;
  offset1 = (t.wxyz >> 24);
  offset2 = t & 0xff;
  offset3 = (t.yzwx >> 8) & 0xff;

  s = ((uint4)(Te2[offset2.x], Te2[offset2.y], Te2[offset2.z], Te2[offset2.w]) & 0x000000ff) ^
    ((uint4)(Te3[offset3.x], Te3[offset3.y], Te3[offset3.z], Te3[offset3.w]) & 0x0000ff00) ^
    ((uint4)(Te0[offset0.x], Te0[offset0.y], Te0[offset0.z], Te0[offset0.w]) & 0x00ff0000) ^
    ((uint4)(Te1[offset1.x], Te1[offset1.y], Te1[offset1.z], Te1[offset1.w]) & 0xff000000) ^
    rk[0];

  state[global_id] = s;
}
```